RL-TR-92-69
In-House Report
April 1992

LARGE-SCALE BATTLEFIELD SIMULATION USING A MULTI-LEVEL MODEL INTEGRATION METHODOLOGY

Alex F. Sisti

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

Rome Laboratory
Air Force Systems Command
Griffiss Air Force Base, NY 13441-5700

This report has been reviewed by the Rome Laboratory Public Affairs Office (PA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

RL-TR-92-69 has been reviewed and is approved for publication.

APPROVED: Thadeus Domurat

THADEUS J. DOMURAT, Chief
Signal Intelligence Division

FOR THE COMMANDER: Garry W. Barringer

GARRY W. BARRINGER
Technical Director
Intelligence & Reconnaissance Directorate

If your address has changed or if you wish to be removed from the Rome Laboratory mailing list, or if the addressee is no longer employed by your organization, please notify RL (IRAE), Griffiss AFB NY 13441-5700. This will assist us in maintaining a current mailing list.

Do not return copies of this report unless contractual obligations or notices on a specific document require that it be returned.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave Blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	April 1992	In-House Nov 91 - Dec 91

4. TITLE AND SUBTITLE
LARGE-SCALE BATTLEFIELD SIMULATION USING A MULTI-LEVEL MODEL INTEGRATION METHODOLOGY

5. FUNDING NUMBERS
PE - 62702F
PR - 4594
TA - I5
WU - I5

6. AUTHOR(S)
Alex F. Sisti

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Rome Laboratory (IRAE)
Griffiss AFB NY 13441-5700

8. PERFORMING ORGANIZATION REPORT NUMBER
RL-TR-92-69

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Rome Laboratory (IRAE)
Griffiss AFB NY 13441-5700

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES
Rome Laboratory Project Engineer: Alex F. Sisti/IRAE/(315)330-4518

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)

This report addresses the problems inherent in the modeling of large-scale and complex software systems in general, and specifically how those problems have affected simulation systems designed to evaluate two particular thrusts in combat simulation: Electronic Combat effectiveness and Non-Cooperative Target Identification (NCTI). Conceptual improvements and potential solutions are offered, leading to an in-depth discussion on a variety of disparate, yet related subject areas. Finally, recommendations are outlined as to future areas of research meriting increased investigation.

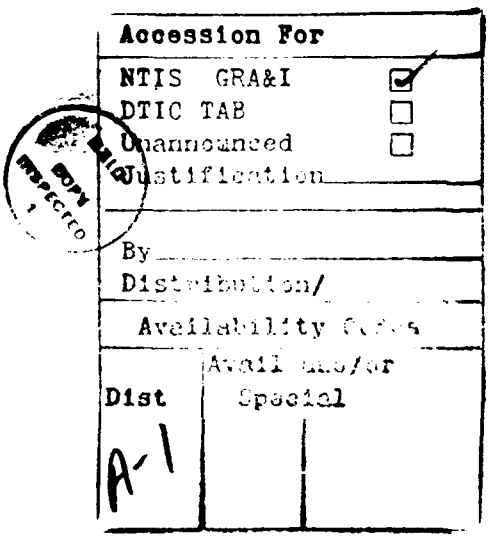
14. SUBJECT TERMS
Simulation, Hierarchy of Models, Model Integration, Software Reuse, "Software Zoom", NCTI Modeling

15. NUMBER OF PAGES
36

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	U/L

PART I: Theoretical Background Issues 1
The Problem 1
An Answer 2
Modularity 2
Software Reuse 4
Hierarchical, Modular Modeling 7
Reuse Systems and Component Coupling 9
Reuse and Modularity Applied to Modeling and Simulation 11
Some Thoughts on Multi-Level Model Integration 12
Some Analogies in the Real World 13
The 'Software Zoom' Concept 15
Another Analogy 15
PART II: Battlefield Simulation (The Concepts are Applied) 16
Validated Analytical Hierarchy of Models (VAHM) 16
Application 1: Electronic Combat Effectiveness 18
A Solution 19
Application 2: Non-Cooperative Target Identification (NCTI) 20
A Solution 21
PART III: Conclusion 22



Summary This report addresses the problems inherent in the modeling of large-scale and complex software systems in general, and specifically how those problems have affected simulation systems designed to evaluate two particular thrusts in combat simulation: Electronic Combat effectiveness and Non-Cooperative Target Identification (NCTI). Conceptual improvements and potential solutions are offered, leading to an in-depth discussion on a variety of disparate, yet related subject areas. Finally, recommendations are outlined as to future areas of research meriting increased investigation.

PART I: Theoretical Background Issues

The Problem

The results of large-scale, monolithic battlefield simulation analyses have generally been coldly received, and with good justification. A scenario of realistic proportions could conceivably involve the modeling of hundreds of thousands of entities, dynamically interacting among themselves, and reacting to other (simulated) activity in their environment. Even given the substantial hardware improvements in the form of larger, faster memories and exponential increases in processing power, it is still impossible to do an analysis at anything less than a grossly aggregated level. It is becoming increasingly imperative that analyses of this sort pay more attention to the underlying details of the entities being modeled; especially since decisions based on these results could ultimately involve human life. As part of a panel discussion at the 1983 Winter Simulation Conference, panel chairman Kenneth Mussleman remarked "Aggregated measures are used to draw conclusions about system performance, while the detailed dynamics of the system go virtually unnoticed ... A decision based solely on summary performance could lead to unacceptable results ... It makes practical sense for us to learn how to properly amplify these details and to incorporate them into the evaluation process." One of the possible alternatives implied by this quote would be to accurately and completely model every entity in the scenario such that its associated details are therefore incorporated into the scenario. This approach is obviously discarded in light of the size and complexity of the simulation, as well as cost, time and resource constraints implied by such an approach.

An Answer

So how then can we model a large-scale, complex software system which would allow us to accurately draw conclusions about detailed performance of a system component or technique within that overall system framework? With the rejection of the extremes (i.e., aggregate analysis based on coarsely represented entities versus completely detailed analyses), what remains is: 1) modeling the complete system from a variety of points of view, or 2) modeling only selected entities/areas of interest or 3) modeling the entire system, following a convention of multiple levels of representation, such that the entities are modeled at varying levels of detail, ranging from the top-level representation of the "essence" of the entity, to the lowest level, which would model the entity in great detail. This clearly has shown the most promise, and brings to bear a variety of technical, theoretical and practical aspects; including modularity, software reuse, object-oriented design, a hierarchy of models in a component library, a model management system for manipulating that library, and software engineering principles in general.

Modularity

Modularity is defined as breaking a software system into smaller and simpler parts or modules such that the modules together perform as the system. In the historical sense, a subroutine is functionally equivalent to a module which can be used repeatedly (and only when desired) in different places in the system. Breaking a large task into smaller, more tractable pieces is certainly not new; it is a time-honored method of increasing productivity in many manufacturing disciplines. It is therefore only natural that the earliest approach away from the traditional monolithic development of software systems was to modularize -- to decompose the system at the functional level and to group related functions together.

Leading the maturation of modularity is the thrust for improvements in the general area of software engineering principles, the foremost being the concepts of data abstraction, encapsulation and information hiding. Data abstraction refers to the process of hiding the implementation details of an object (e.g., program, data) from the users of that object; also called "information hiding" or

"encapsulation". Abstract data types describe classes of objects as a function of their external properties as opposed to their specific computer representation. Deemphasizing a data type's representational details in this way can help eliminate problems of design changes, hardware changes and version compatibility. In other words, as the system undergoes a normal evolution, the actual implementation may be changed without affecting the users of the system. Examples of information likely to be hidden includes peculiarities of the underlying hardware, algorithms to implement a feature specified by the interface, representational details of data structures, and synchronization schemes.

Modular techniques formally began being embodied in programming languages in the mid-1970s with the development of Modula-2, with its "modules", and later, with Ada and its "packages". Still other, more conventional languages can now provide assistance in simulating data abstraction. These modular units contain separate sections for interface specifications and for implementation specifications, thereby supporting data abstraction. Making up the interface section would be information specifying type, data and procedures exported by the module, while the implementation section would contain the executable statements of the interface section, along with locally-used type, data and procedural declarations. Users wishing to invoke such a module to perform its function can access it through the interface section, but consistent with information hiding, cannot see (or affect) how that function has been implemented. The application of data abstraction and of a sister technology thrust, object-oriented design, will be discussed in a later section of this report.

As alluded to earlier, the reasons for modularity are many. A partial list follows:

1. Modularity facilitates writing correct programs. Smaller, more tractable modules, based on sound software engineering principles, are less susceptible to errors.
2. Modular code is easier to design and write. A software design engineer merely has to specify the functional essence of a module in a traditional top-down manner, rather than all of its internal and external details. Actual coding is then facilitated by this visibility of the design structure.
3. A modular approach to a system development allows many programmers to contribute to that development, in parallel. Domain-

specific individual programmers can work on separate functional pieces, guided by the fact that interactions between parts of a system are rigidly restricted to the allowable interactions between those individual pieces.

4. Modularity facilitates easier maintenance as the system evolves. Since almost all large-scale software systems change as the requirements, methodologies or users change, it is essential that a system be easily reconfigurable and maintainable. Since modularity stresses the distinction between implementation changes and interface changes, modifications to the implementation may take place with confidence that inconsistencies will not be introduced to other parts of the system.

5. Modularity facilitates testing, verification and validation. Individual components can be "locally" tested, then fit into the system and re-tested. Furthermore, some applications have exploited modularity to the point of replacing software components with the hardware being simulated.

6. Modularity allows a component hierarchy to be exploited. The concept of a module hierarchy was alluded to earlier in the context of modeling components of a system at varying levels of detail. Bernard Zeigler, who has written many articles on the subject of hierarchical modular modeling [4,7,8,11,36] makes the point that "...models oriented to fundamentally the same objectives may be constructed at different aggregation levels due to tradeoffs in accuracy achievable versus complexity costs incurred." Hierarchical modeling will be discussed in much greater detail in later sections.

7. Lastly, and most important, modularity permits software reuse.

Software Reuse

The process of combining and building up known software elements in different configurations (also called synthesis) has been likened to Gutenberg's concept of removable type. Gutenberg's contribution has historically endured the criticism of purists who argue that his printing press was not so much an invention, as a new application of existing technologies at the time. To that, Douglas McMurtie responded in "The Book", "It does not at all minimize the importance of the invention ... to point out that the invention was the result of a process of synthesis or combination of known elements. For that power of the human mind which can visualize known and familiar facts in new relations, and their

application to new ones -- the creative power of synthesis -- is one of the highest and most exceptional of mental faculties."

The idea of software reuse is certainly not new. Reuse and reworking has been practiced in one form or another since the 1950s; however, the landmark paper in this area is that of M.D. McIlroy's "Mass-Produced Software Components", published in 1968. In that paper, he envisioned and proposed a catalogue of software components from which "software parts could be assembled, much as done with mechanical and electrical components."

Software reuse is defined as the isolation, selection, maintenance and use of software components in the development and maintenance of a software project. Soundly based on the principles of modularity, it has been shown to improve productivity by using previously developed and tested components. Reusable components of a software system include design concepts, functional specifications, algorithms, code, documentation and even personnel. These components embody the various degrees of abstraction which pervade the process of classifying reuse items. Higher degrees of abstraction imply a greater likelihood for reuse. For example, specifications do not yet contain detailed representation details or implementation decisions, so the potential for reuse is greater, while it is very difficult to find pieces of code which can be used without some modifications.

Strictly speaking, software reuse must be distinguished from redesign or reworking. Reuse means using an entity in a different context than what was initially intended, and is also known as "black box" reuse. Redesign or reworking refers to the modification of an existing module before it is used in its new setting. This is known as "white box" reuse, and is by far the more common of the two.

Historically, the classical reusability technique has been to build libraries of routines (e.g., subroutines, functions, procedures), each of which is capable of implementing a well-defined operation. These routines are generally written in a common language for a specific machine, and are accessed by a linker as needed. Although this approach has met with some degree of success in numerical applications, there are some obvious problems which preclude using it

to implement a generally applicable reuse system. Subroutines are too small, representation and implementation details have been filled in, and the glue (interface requirements) necessary to bring many subroutines together is too extensive to make general reuse feasible.

A second approach to reducing or eliminating the software development process takes the form of software generating tools. Software generation has been successfully applied in narrow, well specified domains (e.g., report generators, compiler-compilers, language-based editors), but shows little chance of being used outside these very specific domains. This is primarily because the nature of program generators is such that the application area needs to be very well-defined to achieve the desired level of efficiency.

The third and most promising approach for achieving reusability is based on a technique called object-oriented design. Under object-oriented design, the decomposition of a software system is not based on the functions it performs, but on the classes of objects the system manipulates. Object-oriented programming and languages support the notions of data abstraction, and encapsulation, as described above, and therefore exhibit the flexibility necessary to define and compose reusable components. Some of the more prevalent object-oriented languages include C++, Objective-C, Simula, Smalltalk and some extensions of Pascal and Lisp. In addition, object-oriented principles and constructs are being implemented (or simulated) in other, more conventional languages; especially those which support data abstraction.

Assuming, as many practitioners have, that an object-oriented approach is the best way to describe a software reuse system, many other questions arise. How should existing systems be decomposed? What system fragments are candidates for reuse? How should these candidates be represented and stored? How should they be accessed? Once located, how should they be coupled with other candidate modules, such that together they perform as the system? In the section that follows, these questions are addressed.

Hierarchical, Modular Modeling

Arguably the most prolific of the authors and researchers in the domain of model integration is Bernard P. Zeigler, who first referred to the concept in a 1976 book entitled "Theory of Modeling and Simulation." In that book, he quietly introduced the idea of decomposition of existing models in a hierarchical manner, corresponding to the levels of functional detail. Zeigler [11], discussing the theoretical aspects of decomposition in a hierarchical sense based on varying degrees of detail, states "...specification of design in levels in a hierarchical manner [implies] the first level, and thus the most abstract level, is defined by the behavioral description of the system. Next levels are defined by decomposing the system into subsystems (modules, components) and applying decompositions to such subsystems until the resulting components are judged not to require further decomposition...Therefore, the structure of the specification is a hierarchy where leaf nodes are atomic models (cannot be decomposed any further)." Hierarchical decomposition/representation is fairly extensively treated in the literature; by Zeigler and others.

Zeigler discusses the process of aggregating the details of what he calls base models into "lumped" models. A lumped model includes the functional coverage of one or more detailed component (base) models, modeled with less detail. This involves a simplification process in which the description of the base model is modified in one of the following ways: 1) one or more of the descriptive variables are dropped and their effect accounted for by probabilistic methods, or 2) the range set of one or more of the descriptive variables is coarsened, or 3) similar components are grouped together and their descriptive variables are aggregated. Finally, he pursues a mathematical approach to valid simplification, based on "structure morphisms" at various levels of specification.

In later works, he refined his ideas, and changed the term "lumped" model to coupled model. Most of his contributions to the literature now revolve on his hierarchical, modular composition techniques or actual implementations of his concepts. He says [8] "Considering a real system as a black box, there is a hierarchy of levels at which its models may be constructed ranging from purely behavioral, in which the model claims to represent only the observed input/output behavior of the system, up to the strongly structural, in which

much is claimed about the structure of the system. Simulation models are usually placed at the higher levels of structure and they embody many supposed mechanisms to generate the behavior of interest."

Central to his hierarchical scheme is the composition tree, which describes how components are coupled together to form a composite model. In his words, "Suppose that we have models A and B in the model base. If these model descriptions are in the proper form, then we can create a new model by specifying how the input and output ports of A and B are to be connected to each other and to external ports, an operation called coupling. The resulting model, AB, called a coupled model is once again in modular form ... modularity, as used here, means the description of a model in such a way that it has recognized input and output ports through which all interaction with the external world is mediated. Once in the model base, AB can itself be employed to construct yet larger models in the same manner used with A and B. This property, called closure under coupling, enables hierarchical construction of models." Noting the dual relationship of system decomposition and model synthesis, he remarks that the coupling of two atomic models A and B is associated with the decomposition of the composite model AB into components A and B.

In Zeigler's scheme, which has already been applied to a variety of disciplines, there are three basic parts to the description of an atomic model: 1) the input/output specification, explicitly describing the input and output ports and the ranges their associated variables can assume, 2) the state and auxiliary variables and their ranges (the static structure) and 3) the external and internal transition specification (the dynamic structure). The descriptions for coupled models differs slightly, in that information pertaining to the coupling process is also included. Separate files containing interface specifics are maintained for each component and facilitate the coupling of selected models.

There are three facets to Zeigler's coupling scheme. First, there is a file which contains information relating the input ports of the composite model to the input ports of the components (called external input coupling). Next, external output coupling tells how the output ports of the component model are identified with the output ports of the components. Finally, internal coupling information is maintained, telling how the output ports of the components are connected to

input ports of other components; in other words, it describes how the components inside a composite model are interconnected. Following the principles of abstraction and object-oriented design, all interaction with the environment is mediated through these input and output ports, regardless of the internal implementations of the models. Furthermore, the sending of external events from the output port of one component to the input port of another component can be likened to message passing; another composition technique supported by object-oriented design.

This multifaceted, hierarchical modular modeling concept has been successfully applied in other disciplines; and each time, is improved upon to some degree. The lion's share of his research has been in the representational details of the model base (the component library) and to a limited extent, maintenance of that library -- in essence, a Model Management System (MMS). [Model Management has recently been extensively studied and applied by Benn Konsynski and others [37-44] in relation to its function in a Decision Support System. Essentially, it is an instantiation of a library management system alluded to earlier; providing for the creation, storage, manipulation and accessing of models in a model base. In other words, an MMS is to the Model Base what a DBMS is to a database. It is the cornerstone of a software reuse system, and can be extremely complex in design. Much of the literature in this area confines itself to presenting knowledge representation schemes for implementing a conceptual MMS, but little has been formally done as far as actually building one. Most often, a manual, brute force approach is taken for Model Base population and manipulation. A complete technical discussion of the theory and implementation of Model Management Systems is well beyond the scope of this report; however, the interested reader can find a wealth of information on the subject in the works mentioned above].

Reuse Systems and Component Coupling

The most difficult aspect of the conceptual reuse system deals with coupling, or integrating, existing components together in some manner, to replicate the desired functionality of the system. The benefits of component coupling were discussed in an earlier section of this report. Therefore, this section will address the implementation aspects of component coupling.

Whereas system decomposition is viewed as a classic application of top-down design, component synthesis takes a bottom-up approach. Candidate components meeting the input specifications are put together like building blocks to construct a software system capable of solving the problem. As discussed earlier, these individual components should satisfy the software engineering principles of abstraction, encapsulation and information hiding to reduce the amount of (usually manual) modification needed to integrate them. Software components written in languages which support those features are obviously most desirable from an integration point of view, but that is not to say that software (subroutines, functions etc.) written in a conventional language cannot be used. Again, one of the major design considerations of a reuse system involves possible workarounds because of a desire to retain an expensive or "key" piece of legacy software.

There are two fundamental ways components can be integrated, depending on the degree of interaction required and inter- and intra-dependence of individual components. In the first and simplest case, software modules are run separately and sequentially, using outputs of one component as inputs to the next component to be executed. This method, alternately called chaining or the UNIX pipeline method, is the most straightforward method of integration, and is easily implemented; although some interim transformations or analysis may be required. The second integration method is more complex (often impossible) and is employed when the components are expected to interact with each other. The composition principle used in this case is based on inheritance and message-passing. As expected, the components in a reuse library using this integration method should preferably be those which embrace the policies of object-oriented design, as their interface details are well specified and they can be bound with other components without the internal (implementation) details of the components being known. Object-oriented programming also supports the concepts of object classes and the message-passing between objects. The notion of inheritance is applied in passing messages between classes and subclasses (parents and children). When a new object is sub-classed from a more generic parent class, capabilities common to both are implemented. In addition to being able to process any message that the parent class can (by passing off to the more generic parent), the sub-class can locally process messages which are specific to itself. Again, Ada's "generic packages" and Modula's "modules" meet these

objectives. Another useful construct in Ada and some other object-oriented languages which helps facilitate integration is known as "overloading". Overloading allows more than one meaning to be attached to a name. To use an example from the literature, giving the name "Search" to all associated search procedures enables the user (or the library management system) to always invoke a search operation in the same manner, regardless of the implementation chosen, or the data types. Still another construct being advocated is that of semantic binding. This applies to the flexibility needed when referencing across different domains. The calling component must refer to items it expects in its context, and since it cannot know the items' names in advance, it should be able to refer to them semantically.

One can perhaps begin to see the intricacies involved, and the enormous representational decisions required in developing a reuse system. Reuse systems and their associated tools are costly in terms of time, money and personnel, and even then are often dismissed as unfeasible due to lack of reliable, reusable components. Contractors are hesitant to build software that is too reusable for fear that there may be no "next job" for them. Even the "not invented here" syndrome causes resentment among management and system users. Most importantly, reuse systems have proven to be very application-dependent, with some applications providing a more mature technology base on which to build such a system. One such application area is in support of Modeling and Simulation.

Reuse and Modularity Applied to Modeling and Simulation

Reese and Wyatt [1] speaking at the 1987 Winter Simulation Conference stated "The issues of reusability ... apply to all types of software, including simulation software ... Adoption of a reuse philosophy and the subsequent creation of a reuse library by management is expected to improve the simulation software development process and increase the credibility of simulation results." In addition to the standard components mentioned in earlier sections of this report, there are some support functions which are fairly specific to modeling and simulation, and are required by nearly all discrete simulation applications. Examples of such functions are: time management; queue management; random number generation; data I/O; input sequence front-ends; debug routines;

validation and verification tools; graphical/statistical analysis tools, and animation tools. Obviously, the functions of individual model components are also amenable to reuse technology. In the sections that follow, the discussions on reuse are instantiated to the field of modeling and simulation technology. As part of this instantiation, some of the previously used terms will be changed for clarity. That is, components will be called models; the component library will be designated the Model Base; the library management system will be referred to as the Model Management System (MMS), and component coupling will, in general, be known as model integration.

Some Thoughts on Multi-Level Model Integration

In general, the fields of software reuse and model integration are so new that there is no set formula for deciding what components need to be modeled in finer detail, or how to integrate models of greater fidelity into an already existing software system. Actually, although model reuse and integration present some interesting design problems of their own, the basic issues directly mirror those facing designers of all simulation systems, whose job it is to try to capture the behavior of some real-world process or entity. Briefly, this design phase is characterized by an iterative approach (see Figure 1), consisting of the following steps: Step 1) the real-world process of interest is identified, Step 2) the behavior of the process is roughly modeled, by capturing and coding the knowledge of domain experts, Step 3) that computer model is executed, Step 4) the results go through a process of data reduction and analysis, and Step 5) the fidelity of the model is either increased or decreased. This reworked model then undergoes as many iterations of execution/analysis/modification as necessary; that is, to the point where the domain expert is satisfied with the results of the computer representation of the behavior of the real-world process (Step 6). This digital representation could be anything from a compute-intensive, highly complex interaction of hundreds of input and/or control parameters, to a one-line probabilistic draw.

In either case -- original design and development of a simulation model or the integration (or redesign) of existing models -- the question is essentially the same: What portions of the system need to modeled in

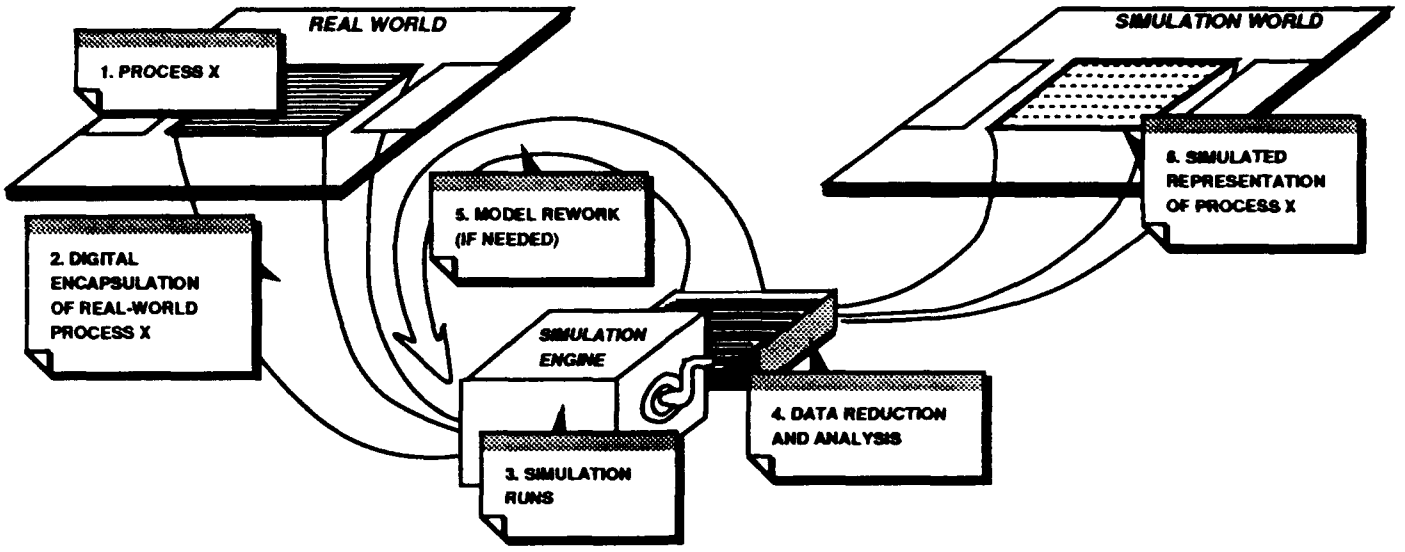


FIGURE 1

detail? Stated most simply, the answer is *those elements which provide the greatest increases in the validity of the simulation results, while imposing the smallest degradation of performance of that simulation.* This implies that there is some "break-even" or "crossover" point(s), arrived at by serious advanced study and planning, including discussions with domain experts (Step 2 above) to determine a) which elements are necessary and/or sufficient, b) which are useful, but whose inclusion might impose a less-than-acceptable performance penalty and c) which elements are just "window-dressing". More than likely, there will need to be a series of independent, statistically-driven experiments performed to converge on the "break-even" point; however, the theory and mechanics of experiment design and interpretation are far beyond the scope of this report. Rather, the remainder of this report will focus on of some of the ideas and lessons learned from past integration efforts, as well as presenting some visual and textual analogies that will hopefully foster a fuller comprehension of model integration.

Some Analogies in the Real World

Unquestionably the easiest way to understand even the most abstract concept is to be able to visualize it; and similarly, the easiest way to visualize a difficult concept is to draw an analogy in the real world, and to map it to the conceptual problem space.

Recalling our conundrum -- the integration of an existing, detailed component into a larger existing system, and how to resolve disparate levels of detail and the interface specifics involved in that integration -- it may be helpful to look at the most complex 'system' ever devised: the human body. As useful and efficient as this system may be, at times parts break down and must be reworked; and sometimes even replaced. We call this operation a transplant, and that is the analogy to be presented here.

When the human system fails to perform efficiently, yielding less-than-satisfactory results, it undergoes a series of tests to isolate the problem. In the case of faulty components, most times they can be fixed. However, in some cases, a component is so irreparably damaged that it no longer adequately serves the purpose for which it was intended. At that point, there begins a series of discussions with the resident domain experts, as to a) whether or not that component is still essential to a satisfactory functioning of the system, b) the 'value-added' to the system, versus the 'cost' of bringing in a replacement component, c) the availability of replacement components, d) the level-of-detail of the replacement component (in essence, a determination of whether to "swap" the existing component with a similar component or to bring in a "new and improved" component), and finally e) given the decision has been made to remove the old component and replace it with the new one, what specific connections need to be made to integrate that component into the system?

This is admittedly a rather ludicrous (and morbid) analogy, but it should shed a little light on the question of 'why integrate?'. As for the question 'how to integrate?', the analogy can again be used: if you cut something during the removal of a component, you'll need to reconnect it (or cap it) when you bring in a replacement. Note that in the case of merely "swapping" similar parts, reconnections are fairly straightforward; it essentially reduces to a one-to-one matchup of all loose ends of the system with those of the replacement component. The problem becomes more difficult when the component to be brought in is dissimilar in size, complexity or function, is built by someone other than the original builders of the system, is based on different specs and assumptions, etc. Ultimately, this kind of analogy points to the need for a standard to which 'spare parts' or improved components should adhere; but again, further discussions along those lines belong in a separate report.

The 'Software Zoom' Concept

As stated earlier, we believe that the best way to model a large scale, complex software system is to model different portions of the system at different levels of detail, and to do such detailed analysis only when needed. A desirable situation would be to build a simulation framework which would allow individual portions of the simulation to be 'toggled' between a course representation of each entity being modeled, and a detailed representation of that entity, to be used only when a closer examination was warranted. Actually, this is not as new an idea as it sounds -- it's as old as the telescope!

Nearly everyone is familiar with the concept of 'zooming'; the act of expanding the view of a specific area of interest. In a similar manner, the act of varying the fidelity of a modeled entity for a more focused look into its workings can be described as a 'software zoom'. It is this ability to replace abstractly modeled entities with more and more detailed models that will ultimately allow validity to be inherited into large-scale software systems.

Another Analogy

The benefits of having the ability to perform a software zoom should be obvious, but how is it accomplished, and what are the implications? Another analogy might be helpful here.

For this analogy, our real world system is the continental United States, and our model of that system is a roadmap. Consider the woman planning a cross-country trip, starting in Los Angeles, ending in New York City, with a 2 day stopover in Topeka, Kansas to visit friends. Her primary goal is to get from the west coast to the east coast, as easily as possible. To successfully accomplish that goal, a roadmap comprised of all the main thoroughfares should be sufficient for most of her trip. However, she may not be familiar with Kansas, or the best way to get from the main highway to the smaller streets leading to her friends' house. Therefore, what she would likely do when she reaches the Kansas state line is to invoke a more detailed representation of that portion of the system -- a Kansas state map -- which would include not only the main thoroughfares, but

also the smaller streets. In fact, when the time is right, she may have yet another model of the system -- a Topeka city map -- which provides the necessary details to bring her to her friends' house. Of course, when she leaves the house, then Topeka and finally, Kansas, the more abstract representation of the real-world system would be sufficient.

What then, in our analogy, is required when 'zooming' in on our traveler's area of interest? Essentially, our removing the 'Kansas' portion of the coarsely modeled system model is somewhat akin to removing the heart in the human system analogy; therefore, according to the formula proposed above, we'd need to determine what connections were severed, and resolve how best to handle their reconnection (or capping). In this case, it reduces to the woman noting what route she is on when she reaches the Kansas state line, locating that route on her Kansas state map, and continuing her journey, using that more refined model.

PART II: Battlefield Simulation: The Concepts are Applied

Validated Analytical Hierarchy of Models (VAHM)

The specific applications of software reuse and model integration of interest to this report involve conducting simulation studies to evaluate combat mission effectiveness. Specifically, we attempt to apply the theoretical concepts discussed in the main body of the report to the problem of modeling the effects of Electronic Combat (EC) in support of mission planning in battlefield scenarios, as well as scenarios to assess the "value added" of Non-Cooperative Target Identification (NCTI) techniques, within a simulated campaign. In addition to the problems which are universally common to software reuse in general, some domain-specific issues are introduced in studies of this sort. In contrast to earlier building block applications of software reuse which, in general, use decomposition schemes based primarily on the implicit functional hierarchy of the system, this study includes another aspect which can be hierarchically described; that being the degree of analysis possible at each level. In a 1979 Air Force study, LCol (then Major) Glen Harris addressed this new aspect in regards to analyzing force effectiveness, and introduced the concept of a "validated analytical hierarchy of models", stating *"Neither a highly detailed*

approach nor a broad aggregate modeling approach by itself is adequate to analyze the complex battlefield. Unless both approaches are used and carefully integrated, the results obtained will not provide the insight required to determine why one ensemble of systems should be preferred over another. An integrated approach must be designed to answer several levels of questions as to the causal relationships involved...". Accordingly, various working groups within the Department of Defense defined a Validated Analytic Hierarchy of Models (VAHM), composed of four levels of analysis; as follows:

Level I: System/Engineering Analysis. The analysis at this level primarily deals with individual systems or components; e.g., jammers, sensors, transmitters, antennas, etc. The objective is to measure the required and/or achieved engineering-level data associated with, for example, Electronic Warfare systems and their interactive effects with target systems [46]. The analysis at this level (and therefore any Measure of Effectiveness associated with this level) is limited to the effects of, in the case of an Electronic Combat scenario, a single jamming component against a single target threat.

Level II: Platform Effects Analysis. At this level, the evaluation focuses on the component being associated with a platform; e.g., a radar jammer installed on an aircraft. The effectiveness of the installed system is then evaluated in the context of a one-on-one or few-on-few analysis.

Level III: Mission Effectiveness Analysis. Analysis at this level assesses the contribution of, for example, Electronic Combat or NCTI techniques to a combat mission environment, including other aspects such as Command and Control, time-sensitive maneuvers, and a defined enemy posture.

Level IV: Force Effectiveness Analysis. This encompasses all the activity associated with operations in the context of joint Air Force/Army/Navy campaigns against an enemy combined arms force, towards evaluating the contribution of, for example, Electronic Combat support in such a campaign.

Under this hierarchical scheme, a distinction is made between vertical integration and horizontal integration. Vertical integration refers to the ability to pass data output of a model at one level of the hierarchy to the input of a

higher (or lower) level model. This is in consonance with the concept of the UNIX pipeline method of integration discussed earlier. Another method of vertical integration is effected by using lower level models as modules in higher level models -- a 'software zoom'. For example, a Level I standard propagation model could be used in a Level II Surface-to-Air weapons model, which could in turn be used in a Level III mission effectiveness model. Vertical integration is important because it provides a validated audit trail of higher-level results to "hard" engineering data, range data, hybrid simulation outputs and/or flight test data. The credibility of most of the upper-level modeling rests on the ability to vertically integrate.

Horizontal integration (federation) of data refers to the accessing (by models at all levels) of a master input/runtime database for data that is global in nature. This concept eliminates the problem of needing multiple databases for multiple models. When intelligence and other situational (state) changes require updating of data, they are changed in one place only, with new values propagating to all models that need to reflect those updates; analogous to the blackboard approach followed in some disciplines of Artificial Intelligence.

Application 1: Electronic Combat Effectiveness

The specific problem which is the subject of this section deals with the lack of fidelity of an existing mission planning tool. Although soundly based on phenomenological and physical properties of barrage jamming (radiated power against radiated power), the tool lacked the required depth, as far as simulating existing Electronic Combat assets to the detail needed for real-life decision making. As in other disciplines, when users required this additional detail, a decision had to be made as to its implementation: rebuild or reuse. For the many reasons described earlier, an integration approach was deemed 1) the most attractive from a cost/time perspective, 2) the most intriguing from a research and development standpoint, but 3) certainly the most difficult (and possibly unattainable), from a realistic point of view.

A Solution

Complexities notwithstanding, an integration approach was pursued. First, existing model repositories were surveyed, as to their compliance with the well-designed criteria of the survey. For instance, each candidate model was compared against such features as model availability, degree of validation, modeling methodology (e.g. stochastic event scheduling versus deterministic scripting), underlying assumptions, Measures of Effectiveness, host software/hardware, dependencies on other models/databases, processing modes (interactive versus batch), availability and currency of documentation, and others too numerous to mention. Finally, three candidate models were chosen, representative of a specific radar jammer, a specific communications jammer and an aircraft dedicated to suppressing enemy air defense systems. These models were obtained from their respective owning/maintaining agencies, and were microscopically (and manually) studied to ascertain and illucidate the necessary integration properties; such as input/output characteristics, units, etc. It should be stressed here that this was a very time- and manpower-intensive undertaking, necessitated in general because of the absence of object-oriented principles in any of the chosen models.

Once modified (minimally, so as not to violate the requirement of maintaining each model's standalone status), the three models were configured to run synchronously -- embodying the 'software zoom' concept -- under the existing simulation executive. Together, the integrated system is called the Electronic Combat Effectiveness System (ECES), and runs on a VAX 11/780. At startup, the three models were informed of the initial conditions of the scenario (e.g., each's own flight path, the perceived threat laydown, geographical/terrain data, etc.), which were, in general, required inputs of each to begin with. In addition, each model contained (or read) its own local data, necessary for standalone execution.

As the ECES model (the aggregate model) is run, messages are sent to the individual (component) models to update positions, announce threat movements and signal activity, and so on. In return, each model passes messages such as flight path changes, jamming noise figures (if requested), or other information pertinent to the mission. The aggregate model serves a dual purpose: it is the

orchestrator of the event script during the scenario (updating the information common to all the models), as well as dynamically displaying all scenario activities (common to all, or as specifically reported by each model) on a graphics terminal.

The Electronic Combat Effectiveness System now runs as it did before its decomposition; the difference with the current system is that validated jammer characteristics are now explicitly modeled, and are inherited (via 'software zoom') as needed. Another major plus is that the synergistic effects of the Electronic Combat assets (taken pairwise or in total) can now be determined. This offers an obvious improvement over the independent execution of the three component models in standalone mode.

During the execution of the system, statistics are collected for ultimate Measures of Effectiveness (MOEs) calculation, corresponding to the four levels of the analysis hierarchy described earlier. For example, in determining mission-level (Level III) effectiveness, a factor called "per cent neutralization" is computed, comparing the ability of (for example) a Target Tracking Radar to successfully track an incoming strike aircraft; both in the presence and absence of jamming support. The actual MOEs and overall results of the asset tradeoffs, while interesting, are beyond the scope of this report. What is of greater interest is the fact that, despite the labor-intensive nature of the integration effort, software reuse was possible, and actually (in this application) facilitated a "total-is-greater-than-the-sum-of-its-parts" system which was not attainable through independent execution of the component models.

Application 2: Non-Cooperative Target Identification (NCTI)

In general, the overall goal of the NCTI program is to assess the feasibility and benefits of improving our aircraft's' long range identification capabilities and techniques, in order to a) better employ beyond-visual- range (BVR) weapons, b) avoid engagement of neutrals, c) reduce fratricide, d) identify enemy-controlled Western-made aircraft and e) better manage and control the air battle. It is immediately obvious that many, if not all, phases of the research and development activity needed for these assessments will involve Modeling and Simulation. Furthermore, it can be assumed that many, many facets of the air

battle will need to be accurately modeled, in order to provide the most realistic assessments possible. Finally, from the discussions above, the conclusion must be drawn that a multi-level model integration approach should be followed.

NCTI simulation activities currently underway in the Air Force involve investigations and analyses at a variety of levels, embodying the ideas of the Validated Analytical Hierarchy of Models discussed earlier. At the Electronic Systems Division (ESD), the NCTI-related tasking of their Project Model office calls for the assessment of NCTI solutions (or as they call it, Hostile Target Identification solutions within a theater-level engagement, complete with all aspects and influences of Command, Control, Communications and Intelligence (C3I) being modeled as well. Rome Laboratory, on the other hand, will initially be investigating proposed advances to existing Electronic Support Measures (ESM) equipment and techniques, and ultimately, developing and configuring models of those advanced capabilities for transition to ESD's Modeling Analysis Simulation Center (MASC) facility.

A Solution

Figure 2 pictorially suggests (without specific details) how ESD might proceed with the integration of an air-to-air engagement model called TAC BRAWLER, into their Level III (theater level) model, TAC SUPPRESSOR. Similarly, Rome Lab will be integrating more detailed Level I (engineering level) components into TAC BRAWLER. Finally, Figure 3 shows how the two organizations are cooperatively working toward the same end.

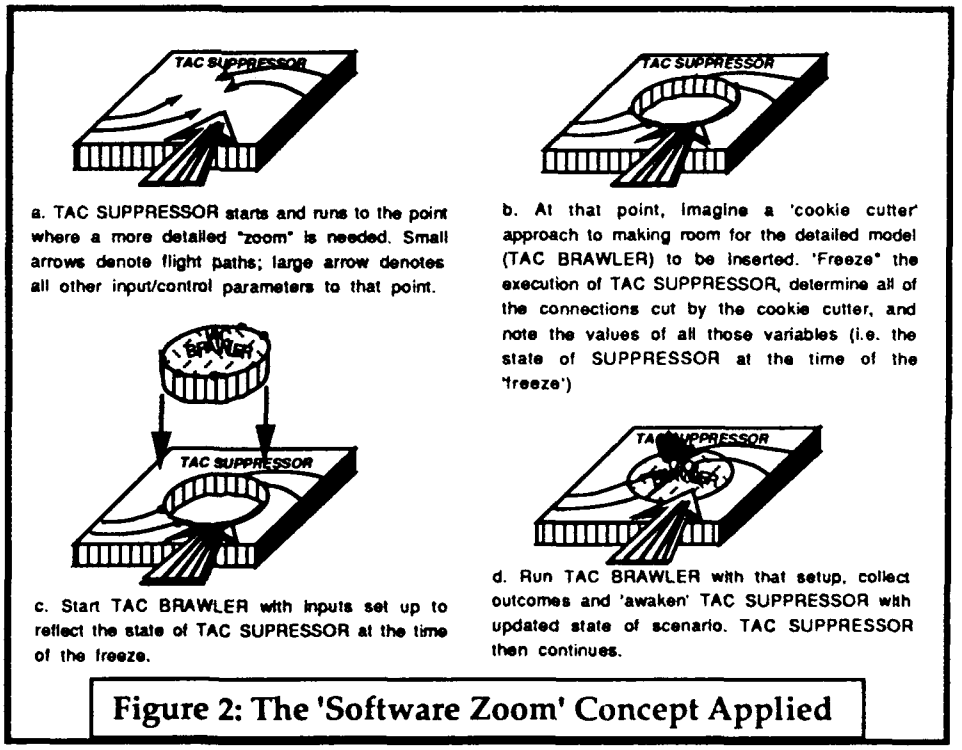


Figure 2: The 'Software Zoom' Concept Applied

PART III: CONCLUSION

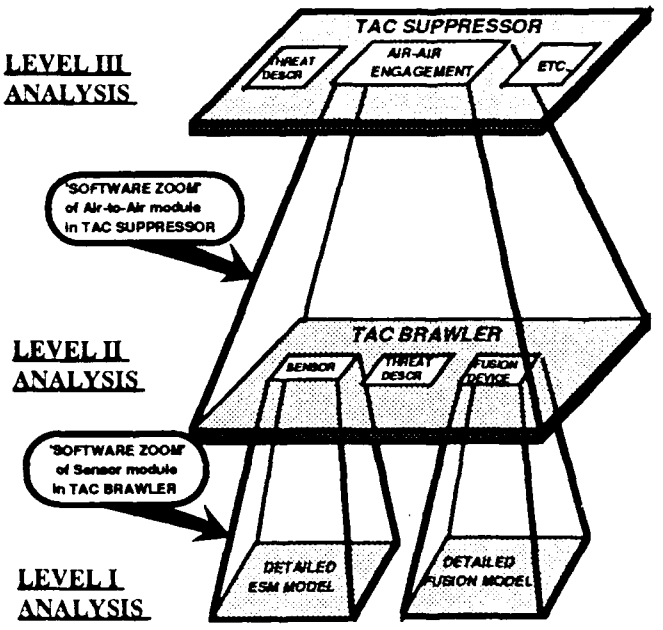


FIG 3: AN EXAMPLE OF MULTI-LEVEL MODEL INTEGRATION WITHIN THE AIR FORCE'S HIERARCHY OF MODEL FRAMEWORK

Despite documented prophesies to the contrary, software reuse and model integration will continue to grow. As expected improvements in programming practices and standardization begin to materialize, reuse system development will migrate from this awkward period of "working with the givens", towards the inception of standard models, interfaces, and perhaps even packaged standard libraries. This next generation of software system development will not be without cost. There will be great outlays and sacrifices in all areas; not the least of which will be obtaining management support. But no matter the cost, no matter the effort, no matter the level of resistance encountered, this technology area should be vigorously pursued, as were many other unlikely, yet promising areas, which are now standard practices. Simply put: In order to introduce validity -- and therefore acceptance -- to a large-scale, theater level battlefield simulation, there is no other alternative.

BIBLIOGRAPHY

1. R. Reese, D. L. Wyatt, "Software Reuse and Simulation", Proceedings of the 1987 Winter Simulation Conference

2. G. C. Vansteenkiste, "New Challenges in System Simulation", Proceedings of the 1985 Summer Computer Simulation Conference

3. K. J. Murray, S. V. Sheppard, "Automated Model Synthesis: Using Automatic Programming and Expert Systems Techniques Toward Simulation Modeling", Proceedings of the 1987 Winter Computer Simulation Conference

4. B. P. Zeigler, T. G. Kim, "The DEVS Formalism: Hierarchical, Modular Systems Specification in an Object Oriented Framework", Proceedings of the 1987 Winter Computer Simulation Conference

5. D. Kostelski, J. Buzacott, K. McKay, X. Liu, "Development and Validation of a System Macro Model Using Isolated Micro Models", Proceedings of the 1987 Winter Computer Simulation Conference

6. R. G. Sargent, "An Overview of Verification and Validation of Simulation Models", Proceedings of the 1987 Winter Computer Simulation Conference

7. B. P. Zeigler, "Hierarchical Modular Modeling/Knowledge Representation", Proceedings of the 1987 Winter Computer Simulation Conference

8. B. P. Zeigler, T. I. Oren, "Multifaceted, Multiparadigm Modelling Perspectives: Tools for the 90s", Proceedings of the 1987 Winter Computer Simulation Conference

9. R. G. Sargent, "Joining Existing Simulation Programs", Proceedings of the 1987 Winter Computer Simulation Conference

10. A. I. Concepcion, S. J. Schon, "SAM - A Computer Aided Design Tool for Specifying and Analyzing Modular, Hierarchical Systems", Proceedings of the 1987 Winter Computer Simulation Conference

11. J. W. Rozenblit, S. Sevinc, B. P. Zeigler, "Knowledge-Based Design of LANs Using System Entity Structure Concepts", Proceedings of the 1987 Winter Computer Simulation Conference

12. S. A. Shoaf, "A Modular Approach to the Simulation of Manufacturing Processes", Proceedings of the 1983 Winter Computer Simulation Conference

13. K. J. Musselman, et al, "Practitioners' Views on Simulation", panel discussion from the Proceedings of the 1983 Winter Computer Simulation Conference

14. W. T. Jones, B. J. Jones, "Computer Simulation Using Hierarchical Models", Proceedings of the 6th Pittsburgh Conference, 1975

15. B. R. Konsynski, J. F. Nunamaker, "A Generalized Model for Computer-Aided Process Organization in Design of Information Systems", Proceedings of the 6th Pittsburgh Conference, 1975

16. D. W. Balmer, "Modeling Styles and Their Support in the CASM Environment", Proceedings of the 1987 Winter Computer Simulation Conference

17. R. Prieto-Diaz, P. Freeman, "Classifying Software for Reusability", IEEE Software, Jan 87, p6

18. R. F. Kamel, "Effect of Modularity on System Evolution", IEEE Software, Jan 87, p48

19. A. Reilly, "Roots of Reuse", IEEE Software, Jan 87, p4

20. B. D. Shriver, "Reuse Revisited", IEEE Software, Jan 87, p5

21. S. Chang, "Visual Languages: A Tutorial and Survey", IEEE Software, Jan 87, p29

22. W. Tracz, "Reusability Comes of Age", IEEE Software, Jan 87, p6

23. P. G. Bassett, "Frame-Based Software Engineering", IEEE Software, Jan 87, p9

24. G. E. Kaiser, D. Garlan, "Melding Software Systems from Reusable Building Blocks", IEEE Software, Jan 87, p17

25. B. A. Burton et al, "The Reusable Software Library", IEEE Software, Jan 87, p25

26. M. Lenz, H. A. Schmid, P. F. Wolf, "Software Reuse Through Building Blocks", IEEE Software, Jan 87, p34

27. A. Gargaro, T. L. Pappas, "Reusability Issues and Ada", IEEE Software, Jan 87, p43

28. S. N. Woodfield, D. W. Embley, D. T. Scott, "Can Programmers Reuse Software?", IEEE Software, Jan 87, p52

29. G. Fischer, "Cognitive View of Reuse and Redesign", IEEE Software, Jan 87, p60

30. R. Conn, "ADA Software Repository", IEEE Software, Jan 87, p105

31. T. Biggerstaff, C. Richter, "Reusability Framework, Assessment, and Directions", IEEE Software, Mar 87, p41

32. B. Meyer, "Reusability: The Case for Object-Oriented Design", IEEE Software, Mar 87, p50

33. K. W. Miller, L. J. Morell, F. Stevens, "Adding Data Abstraction to Fortran Software", IEEE Software, Nov 88, p50

34. G. Gruman, "Early Reuse Lives Up To Its Promise", IEEE Software, Nov 88, p87

35. J. R. Emshoff, R. L. Sisson, "Design and Use of Simulation Models"

36. B. P. Zeigler, "Theory of Modeling and Simulation"

37. Dolk, B. Konsynski, "Knowledge Representation for Model Management Systems",

38. O. I. Truncer, "Concepts and Criteria to Assess Acceptability of Simulation Studies: A Frame of Reference", *Communications* of the ACM, Vol 24, No. 4, Apr 81

39. Klein, Konsynski, Beck, "A Linear Representation for Model Management in a DSS", Journal of Management Information Systems, Vol II, No 2, Fall 85

40. McIntyre, Konsynski, Nunamaker, Jr., "Automating Planning Environments: Knowledge Integration and Model Scripting", Journal of Management Information Systems, Vol II, No 4, Spring 86

41. Liang, Ting-Peng, "Integrating Model Management with Data Management in DSS", Decision Support Systems 1 (1985), p221

42. Applegate, Konsynski, Nunamaker, Jr., "Model Management Systems: Design for Decision Support", Decision Support Systems 2 (1986), p81

43. Konsynski, Sprague, "Future Research Directions in Model Management", Decision Support Systems 2 (1986), p 103

44. Fedorowicz, Williams, "Representing Modeling Knowledge in an Intelligent Decision Support System", Decision Support Systems 2 (1986), p 3

45. G. L. Harris, "Computer Models, Laboratory Simulators and Test Ranges: Meeting the Challenge of Estimating Tactical Force Effectiveness in the 1980s", 1979

46. G. R. Dougherty, "On What Basis, EW?", Journal of Electronic Defense, Oct 84

47. A. F. Sisti, et al, "Electronic Combat Development and Demonstration Component", RADC-TM-86-9, Aug 86, B104997.

48. A. F. Sisti, et al, "Automated Intelligence Decision Aids", RADC-TR-87-17, Feb 87, B109815.

49. A. F. Sisti, "A Model Integration Approach to Electronic Combat Effectiveness Evaluation", RADC-TR-89-183, Oct 89, A215804.

50. M. Ringler and G. Brown, "Electronic Combat Effectiveness Study", RADC-TR-88-276, Nov 88, B132176L.

MISSION

OF

ROME LABORATORY

Rome Laboratory plans and executes an interdisciplinary program in research, development, test, and technology transition in support of Air Force Command, Control, Communications and Intelligence (C^3I) activities for all Air Force platforms. It also executes selected acquisition programs in several areas of expertise. Technical and engineering support within areas of competence is provided to ESD Program Offices (POs) and other ESD elements to perform effective acquisition of C^3I systems. In addition, Rome Laboratory's technology supports other AFSC Product Divisions, the Air Force user community, and other DOD and non-DOD agencies. Rome Laboratory maintains technical competence and research programs in areas including, but not limited to, communications, command and control, battle management, intelligence information processing, computational sciences and software producibility, wide area surveillance/sensors, signal processing, solid state sciences, photonics, electromagnetic technology, superconductivity, and electronic reliability/maintainability and testability.